Chapter 1

Up from the Primordial C

In This Chapter

- Hysterical C history
- How C programs are created
- Building the source code
- Compiling and linking
- Running the result

As the most useful device you have ever used, a computer can become anything — as long as you have the ability to program it. That's what makes computers unique in the pantheon of modern devices. And although most computer users shy away from programming — confusing it with mathematics or electrical engineering — the fact is that programming a computer is really a rather simple and straightforward thing. It's easy.

This chapter introduces you to the basics of programming. Although it has some yabber-yabber and background information, the meat of the chapter involves creating, compiling, and running your first program. Feel the power! Finally, it's *you* who can tell the computer what to do with itself!

Because you probably didn't read this book's Introduction (for shame), know that you should preview Appendix A before starting here.

An Extremely Short and Cheap History of the C Language

First, there was the B programming language. Then there was the C programming language.

Stuff you don't need to know about language levels

Programming languages have different levels, depending on how much they resemble human languages. Programming languages that use common words and are relatively easy for most folks to read and study are called *high-level* languages. The opposite of those are *low-level* languages, which are not easy to read or study.

High-level languages include the popular BASIC programming language as well as other languages that just aren't that popular any more. BASIC reads almost like English, and all its commands and instructions are English words — or at least English words missing a few vowels or severely disobeying the laws of spelling.

The lowest of the low-level programming languages is machine language. That language is the actual primitive grunts and groans of the microprocessor itself. Machine language consists of numbers and codes that the microprocessor understands and executes. Therefore, no one really writes programs in machine language; rather, they use assembly language, which is one step above the low-level machine language because the grunts and groans are spelled out rather than entered as raw numbers.

Why would anyone use a low-level language when high-level languages exist? Speed! Programs written in low-level languages run as fast as the computer can run them, often many times faster than their high-level counterparts. Plus, the size of the program is smaller. A program written in Visual Basic may be 34K in size, but the same program written in assembly language may be 896 bytes long. On the other hand, the time it takes to develop an assembly language program is much longer than it would take to write the same program in a higher-level language. It's a trade-off.

The C programming language is considered a mid-level language. It has parts that are low-level grunting and squawking, and also many high-level parts that read like any sentence in a Michael Crichton novel, but with more character development. In C, you get the best of the high-level programming languages and the speed of development they offer, and you also get the compact program size and speed of a low-level language. That's why C is so bitchen.

No, I'm not being flip. C was developed at AT&T Bell Labs in the early 1970s. At the time, Bell Labs had a programming language named B — B for Bell. The next language they created was C — one up on B.

- C is the offspring of both the B programming language and a language named BCPL, which stood for Basic Combined Programming Language. But you have to admit that the B story is cute enough by itself.
- You would think that the next, better version of C would be called the D language. But, no; it's named C++, for reasons that become apparent in Chapter 16.
- C is considered a *mid-level language*. See the nearby sidebar, "Stuff you don't need to know about language levels," for the boring details.

- The guy who created the C programming language at Bell Labs is Dennis Ritchie. I mention him in case you're ever walking on the street and you happen to bump into Mr. Ritchie. In that case, you can say "Hey, aren't you Dennis Ritchie, the guy who invented C?" And he'll say "Why — why, yes I am." And you can say "Cool."

The C Development Cycle

Here is how you create a C program in seven steps — in what's known as the *development cycle:*

1. Come up with an idea for a program.
2. Use an editor to write the source code.
3. Compile the source code and link the program by using the C compiler.
4. Weep bitterly over errors (optional).
5. Run the program and test it.
6. Pull out hair over bugs (optional).
7. Start over (required).

No need to memorize this list. It's like the instructions on a shampoo bottle, though you don't have to be naked and wet to program a computer. Eventually, just like shampooing, you start following these steps without thinking about it. No need to memorize anything.

- The C development cycle is not an exercise device. In fact, programming does more to make your butt fit more snugly into your chair than anything.
- Step 1 is the hardest. The rest fall naturally into place.
- Step 3 consists of two steps: compiling and linking. For most of this book, however, they are done together, in one step. Only later — if you're still interested — do I go into the specific differences of a compiler and a linker.

From Text File to Program

When you create a program, you become a programmer. Your friends or relatives may refer to you as a "computer wizard" or "guru," but trust me when I say that *programmer* is a far better title.

As a programmer, you job is not "programming." No, the act of writing a program is *coding*. So what you do when you sit down to write that program is *code* the program. Get used to that term! It's very trendy.

The job of the programmer is to write some code! Code to do what? And what type of code do you use? Secret code? Morse Code? Zip code?

The purpose of a computer program is to make the computer do something.

The object of programming is to "make it happen." The C language is only a tool for communicating with the PC. As the programmer, it's your job to translate the intentions of the computer user into something the computer understands and then give users what they want. And if you can't give them what they want, at least make it close enough so that they don't constantly complain or — worse — want their money back.

The tool you have chosen to make it happen is the C programming language. That's the code you use to communicate with the PC. The following sections describe how the process works. After all, you can just pick up the mouse and say "Hello, computer!"

- Programming is what TV network executives do. Computer programmers *code*.
- You use a programming language to communicate with the computer, telling it exactly what to do.

The source code (text file)

Because the computer can't understand speech and, well, whacking the computer — no matter how emotionally validating that is for you — does little to the PC, your best line of communications is to write the computer a note — a file on disk.

To create a PC epistle, you use a program called a text editor. This program is a primitive version of a word processor minus all the fancy formatting and printing controls. The text editor lets you type text — that's about all.

Using your text editor, you create what's called a *source code file.* The only special thing about this file is that it contains instructions that tell the computer what to do. And although it would be nice to write instructions like "Make a funny noise," the truth is that you must write instructions in a tongue the computer understands. In this case, the instructions are written in the C language.

- The source code file is a text file on disk. The file contains instructions for the computer that are written in the C programming language.
- You use a text editor to create the source code file. See Appendix A for more information on text editors.

Creating the GOODBYE.C source code file

Use your text editor to create the following source code. Carefully type each line exactly as written; *everything* you see below is important and necessary. Don't leave anything out:

```
#include <stdio.h>

int main()
{
    printf("Goodbye, cruel world!\n");
    return(0);
}
```

As you review what you have typed, note how much of it is familiar to you. You recognize some words (`include`, `main`, `"Goodbye, cruel world!"`, and `return`), and some words look strange to you (`stdio.h`, `printf`, and that `\n` thing).

When you have finished writing the instructions, save them in a file on disk. Name the file GOODBYE.C. Use the commands in your text editor to save this file, and then return to the command prompt to compile your instructions into a program.

- See Appendix A for information on using a text editor to write C language programs as well as for instructions on where you should save the source code file on disk.
- In Windows Notepad, you must ensure that the file ends in `.C` and not in `.TXT`. Find a book about Windows for instructions on showing the filename extensions, which makes saving a text file to disk with a .C extension easier.
- Note that the text is mostly in lowercase. It must be; programming languages are more than case sensitive — they're case-*fussy*. Don't worry when English grammar or punctuation rules go wacky; C is a *computer* language, not English.
- Also note how the program makes use of various parentheses: the angle brackets, `<` and `>`; the curly braces, `{` and `}`; and the regular parentheses, `(` and `)`.

Extra help in typing the GOODBYE.C source code

The first line looks like this:

```
#include <stdio.h>
```

Type a pound sign (press Shift+#) and then **include** and a space. Type a left angle bracket (it's above the comma key) and then **stdio**, a period, **h**, and a right angle bracket. Everything must be in lowercase — no capitals! Press Enter to end this line and start the second line.

Press the Enter key alone on the second line to make it blank. Blank lines are common in programming code; they add space that separates pieces of the code and makes it more readable. And, trust me, anything that makes programming code more readable is okay by me!

Type the word **int**, a space, **main**, and then two parentheses hugging nothing:

```
int main()
```

There is no space between `main` and the parentheses and no space inside the parentheses. Press Enter to start the fourth line.

Type a left curly brace:

```
{
```

This character is on a line by itself, right at the start of the line. Press Enter to start the fifth line.

```
printf("Goodbye, cruel
   world!\n");
```

If your editor was smart enough to automatically indent this line, great. If not, press the Tab key to indent. Then type **printf**, the word *print* with a little *f* at the end. (It's pronounced "print-eff.") Type a left parenthesis. Type a double quote. Type **Goodbye, cruel world**, followed by an exclamation point. Then type a backslash, a little **n**, double quotes, a right parenthesis, and, finally, a semicolon. Press Enter to start the sixth line.

```
return(0);
```

If the editor doesn't automatically indent the sixth line, press the Tab key to start the line with an indent. Then type **return**, a paren, **0** (zero), a paren, and a semicolon. Press Enter.

On the seventh line, type the right curly brace:

```
}
```

Some editors automatically unindent this brace for you. If not, use your editor to back up the brace so that it's in the first column. Press the Enter key to end this line.

Leave the eighth line blank.

The compiler and the linker

After the source code is created and saved to disk, it must be translated into a language the computer can understand. This job is tackled by the compiler.

The *compiler* is a special program that reads the instructions stored in the source code file, examines each instruction, and then translates the information into the machine code understood only by the computer's microprocessor.

If all goes well and the compiler is duly pleased with your source code, the compiler creates an object code file. It's a middle step, one that isn't necessary for smaller programs but that becomes vital for larger programs.

Finally, the compiler links the object code file, which creates a real, live computer program.

If either the compiler or the linker doesn't understand something, an error message is displayed. At that point, you can gnash your teeth and sit and stew. Then go back and edit the source code file again, fixing whatever error the compiler found. (It isn't as tough as it sounds.) Then you attempt to compile the program again — you recompile and relink.

- The compiler translates the information in the source code file into instructions the computer can understand. The linker then converts that information into a runnable program.
- The GCC compiler recommended and used in this book combines the compiling and linking steps. An object file *is* created by GCC, but it is automatically deleted when the final program file is created.
- Object code files end in OBJ or sometimes just O. The first part of the object file name is the same as the source code filename.
- Feel free to cheerfully forget all this object code nonsense for now.
- Text editor➪Compiler.
- Source code➪Program.

Compiling GOODBYE.C

The gritty details for compiling a program are in Appendix A. Assuming that you have thumbed through it already, use your powerful human memory to recall the proper command to compile and link the GOODBYE.C source code. Here's a hint:

```
gcc goodbye.c -o goodbye
```

Type that command at your command prompt and see what happens.

Well?

Nothing happens! If you have done everything properly, the GCC compiler merely creates the final program file for you. The only time you see a message is if you goof up and an error occurs in the program.

If you do get an error, you most likely either made a typo or forgot some tiny tidbit of a character: a missing " or ; or \ or) or (or — you get the idea. *Very carefully* review the source code earlier in this chapter and compare it with what you have written. Use the editor to fix your mistake, save the code to disk, and then try again.

Note that GCC reports errors by line number, or it may even specifically list the foul word it found. In any event, note that Chapter 2 covers error-hunting in your C programs.

Running the final result

If you used the proper compiling command, the name of the program to run is identical to the first part of your source code. So why not run that program!

In Windows, the command to type is

```
goodbye
```

In the Unix-like operating systems, you must specify the program's path or location before the program name. Type this command:

```
./goodbye
```

Press the Enter key and the program runs, displaying this marvelous text on your screen:

```
Goodbye, cruel world!
```

Welcome to C language programming!

(See Appendix A for more information on running programs.)

Save It! Compile and Link It! Run It!

Four steps are required in order to build any program in C. They are save, compile, link, and run. Most C programming language packages automatically perform the linking step, though whether or not it's done manually, it's still in there.

Save! Saving means to save your source code. You create that source code in a text editor and save it as a text file with the C (single letter C) extension.

Compile and link! Compiling is the process of transforming the instructions in the text file into instructions the computer's microprocessor can understand. The linking step is where the instructions are finally transformed into a program file. (Again, your compiler may do this step automatically.)

Run! Finally, you run the program you have created. Yes, it's a legitimate program, like any other on your hard drive.

You have completed all these steps in this chapter, culminating in the creation of the GOODBYE program. That's how C programs are built. At this stage, the hardest part is knowing what to put in the source file, which gets easier as you progress through this book. (But by then, getting your program to run correctly and without errors is the hardest part!)

You find the instructions to save, compile, and run often in this book. That's because these steps are more or less mechanical. What's more important is understanding how the language works. That's what you start to find out about in the next chapter.